Examensarbete, Högskolan på Åland, Utbildningsprogrammet för Informationsteknik

OMSTRUKTURERING AV LEGACY-KOD MED HJÄLP AV DOMÄNDRIVEN DESIGN

Philip Eriksson

18:2018

Datum för godkännande: 22.05.2018
Handledare: Agenta Eriksson-Granskog

EXAMENSARBETE
Högskolan på Åland

Utbildningsprogram:	Informationsteknik
Författare:	Eriksson, Philip
Arbetets namn:	Omstrukturering av legacy-kod med hjälp av domändriven design
Handledare:	Agneta Eriksson-Granskog
Uppdragsgivare:	Crosskey Banking Solutions

Abstrakt

Syftet med detta arbete är att bygga om en del av avgiftshanteringen inom Crosskeys kortsystem. Idag är denna hantering en del av en växande legacy-kodbas, vilket leder till att ändringar och underhåll av avgifter ofta blir en komplicerad och tidskrävande process.

I arbetet ingår att driva systemet till en mera domändriven design, där de olika delarna av systemet är fristående ifrån varandra, för att minska på beroenden.

I arbetet presenteras vår bild av domändriven design och hur vi planerar att använda de principerna, samt kortfattat om vad legacy-kod innebär och hur man jobbar med det.

Nyckelord (sökord)

Java, Spring, Domändriven Design (DDD), Legacy, Avgifter, Omstrukturering

Högskolans serienummer:	ISSN:	Språk:	Sidantal:
18:2018	1458-1531	Svenska	25 sidor

Inlämningsdatum:	Presentationsdatum:	Datum för godkännande:
14.5.2018	16.5.2018	22.5.2018

DEGREE THESIS
Åland Univeristy of Applied Sciences

Study program:	Information Technology
Författare:	Eriksson, Philip
Arbetets namn:	Refactoring of Legacy Code using Domain Driven Design
Handledare:	Agneta Eriksson-Granskog
Uppdragsgivare:	Crosskey Banking Solutions

Abstract

The purpose of this thesis is to rebuild part of the fee handling in Crosskeys card system. Today this handling is a part of a growing legacy codebase, which makes changes and maintenance of fees into a complex and time-consuming process.

The thesis includes trying to drive the system towards a more domain driven approach, where the different parts of the system are independent of eachother.

In the thesis I present our approach to domain driven design and how we plan on using those principles, along with a section about legacy-code and how to work with it.

Keywords

Java, Spring, Domaindriven Design (DDD), Legacy, Fees, Refactoring

Serial Number	ISSN:	Language:	Number of Pages:
18:2018	1458-1531	Swedish	25 pages

Handed in:	Date of presentation:	Date of approval:
14.5.2018	16.5.2018	22.5.2018

INNEHÅLLSFÖRTECKNING

1. Introduktion

1.1 Syfte

Syftet med detta arbete är att omstrukturera en viktig del av beställarens kodbas, nämligen deras avgiftshantering. I dagsläget skrivs ny kod oftast i egna moduler, för att separera på ansvarsområden, enligt goda objektorienterade principer. Men en stor del av vår kod ligger fortfarande i ett och samma bibliotek, utan någon klar separation av ansvarsområden.

1.2 Metod

Till en början var kravspecifikationen för arbetet väldigt lång. Slutsatsen vi drog var att arbetet skulle bestå främst av förberedande delar av ett större projekt. Kunskapen för att utföra kommer i stor utsträckning från Högskolan på Åland, men även genom kontakt och genomgångar på Crosskey.

Omstruktureringen kommer att göras i flera steg, som kommar att dokumenteras närmare senare i arbetet. Målet är att driva hela kodbasen mot en mera Domändriven Design (DDD), för att enklare kunna göra ändringar i systemet och minska ner på beroenden mellan moduler.

Så mycket som möjligt kommer att göras om från grunden, men för den mer komplexa logiken så kommer implementationen att vara den samma som tidigare.

Arbetet kommer att ske med fria händer vad gäller design och val av tekniker, med regelbundna samtal och frågeställningar till min handledare på företaget.

1.3 Avgränsningar

En komplett DDD-struktur över systemet skulle ta för lång tid för detta arbete, så vi valde att begränsa arbetet till att omstrukturera den del av koden som ansvarar för att en särskild avgift ska postas till ett konto.

Några viktiga delar som uteblev ur arbetet p.g.a. tidsbrist var ändringar i hur avgifter hanteras i vårt backoffice-system.

1.4 Beställare

Crosskey Banking Solutions är ett dotterbolag till Ålandsbanken, som erbjuder utveckling och underhåll av IT-system inom bank- och finanssektorn. Crosskey har upp emot 240 anställda på flera orter, med huvudkontoret i Mariehamn. (Crosskey, 2018)

Arbetet har gjorts inom avdelningen Cards & Mobile Payments.

Figur 1: Crosskey Banking Solutions

2. BAKGRUND / TEKNIKER

2.1 Domändriven design (DDD)

> *DDD isn't first and foremost about technology.*
> *In its most central principles, DDD is about discussion, listening, understanding, discovery, and business value, all in an effort to centralize knowledge.* (Vernon, 2013)

Domändriven design (förkortat till DDD) handlar, på teknisk nivå, om att koden man skriver följer affärsområdets modell. Vad det innebär är att verksamhetens bild av systemet stämmer överens med implementationen. Detta gör det inte bara lättare för nya utvecklare att förstå systemet, utan underlättar även kommunikation mellan utvecklare och annan personal i teamen.

I detta arbete kommer DDD att användas för att separera på applikationens logik i fristående domäner. Följer man principerna till fullo ska all affärslogik finnas i domänobjekt och services. Vår tanke är att varje domän ska ansvara för en del av applikationen, utan att överlappa till andra domäner, för att hålla antalet beroenden lågt som möjligt.

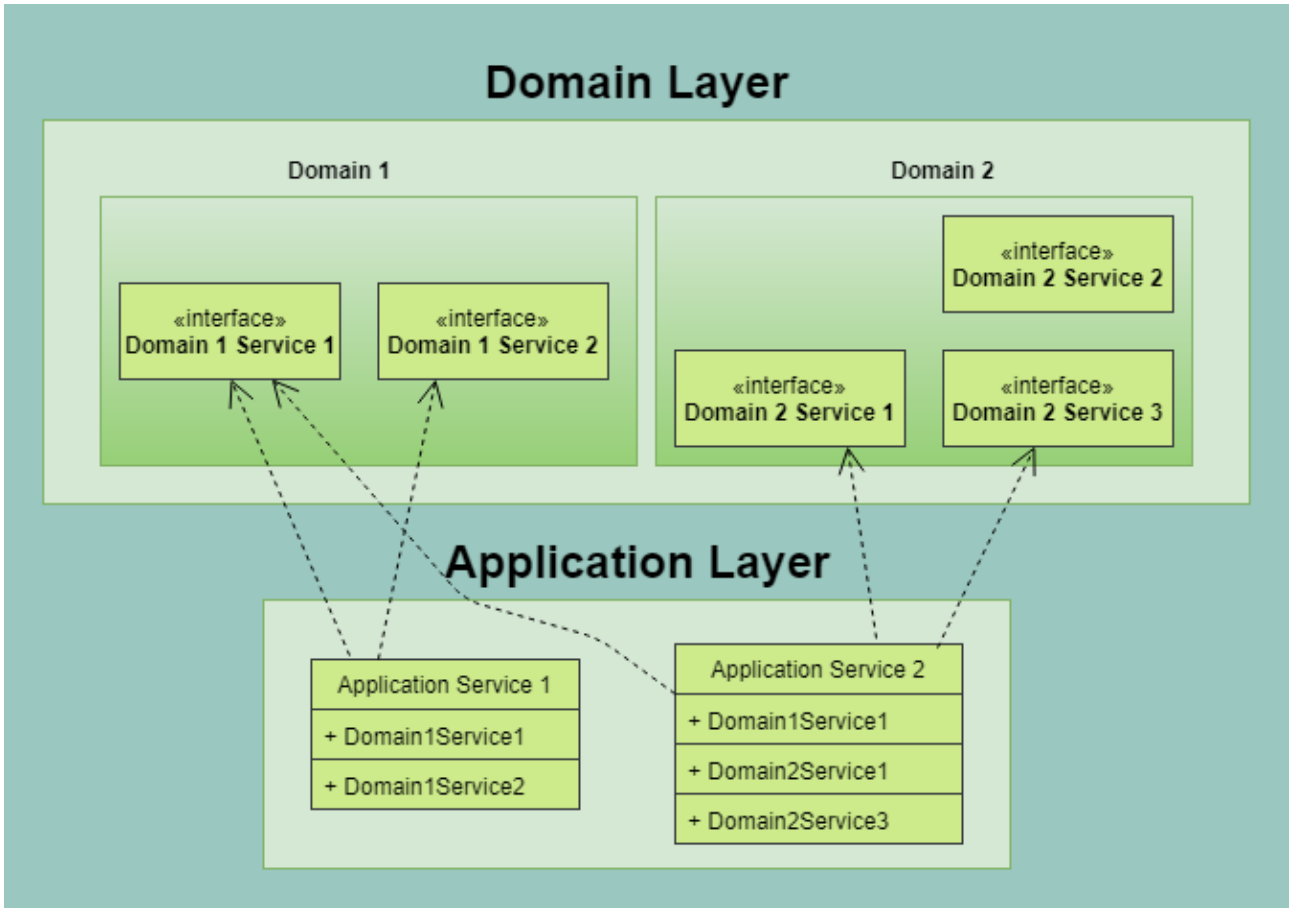
En komplett DDD-struktur över systemet skulle kräva att flera delar av systemet följer DDD-principerna, så istället använder vi ett slags mellansteg, där vi delar upp systemet i domän- och applikationslager. På så sätt kan vi flytta så mycket logik som möjligt till domänerna, och sedan ”knyta ihop” dem med hjälp av service-klasser i applikationslagret. I figur 2 ses en bild på hur en sådan uppdelning kan se ut.

Vår användning av DDD följer inte design-principerna fullt ut, eftersom en sådan struktur skulle kräva en stor insats av ett flertal människor inom flera olika delar av applikationen. Därför har vi bestämt att endast implementera några av principerna för DDD.

Vår plan är att dela upp koden i två lager:

- Ett domänlager där enskilda, fristående domäner har ansvar för sin del av applikationen, utan att ha några som helst beroenden till andra domäner.
- Ett applikationslager där vi kan ”knyta ihop” flera olika domäner.

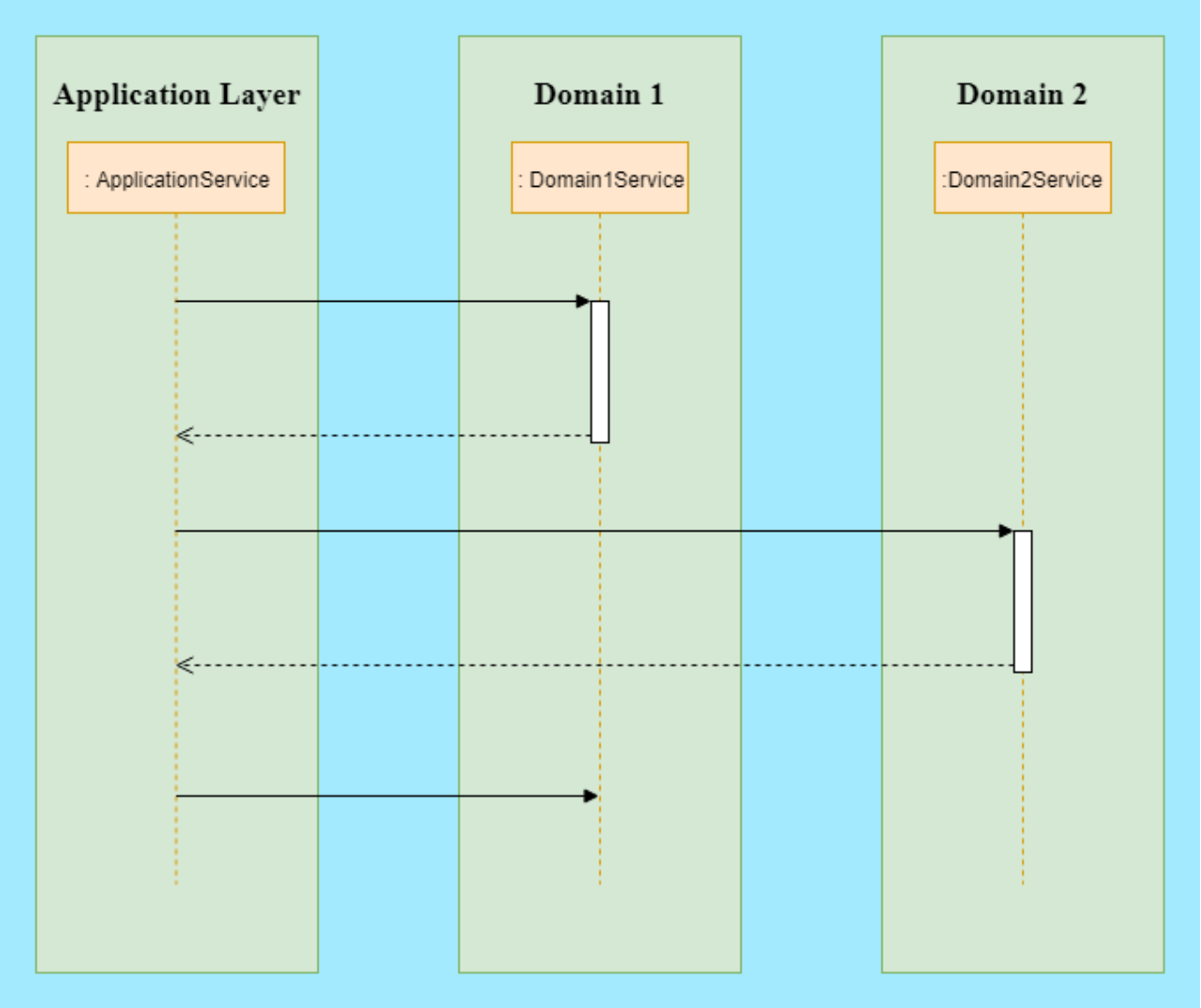
En sådan uppdelning leder till bättre förståelse om vilken logik som ska vara var, samt att det blir lättare att visualisera och hantera beroenden mellan olika delar av systemet. Figur 2 visar hur de två lagren kan kommunicera med varandra.



Figur 2: Modell över domän- och applikationslager

I figuren ser vi att applikationslagret binder samman två (eller flera) domäner, utan att domänerna behöver kommunicera med varandra. På så vis blir det lättare att med tiden flytta mer och mer logik från applikationslagret till domänlagret. Applikationslagret kommer att agera som en startpunkt för en kedja av metodanrop och kommer att börja från någon slags användargränssnitt eller schemalagda jobb.

Ett exempel på en sådan kedja visas i följande sekvensdiagram i Figur 3:



Figur 3: Sekvensdiagram för ett anrop till applikations-servicen

2.2 Java

Java är ett objektorienterat programmeringsspråk som stöder flera olika ramverk och externa bibliotek. Java följer ”WORA”-principen (Write Once, Run Anywhere), vilket betyder att Java-kod som är kompilerad kan köra på alla enheter som har en Java Virtual Machine installerad. (Perry, 2017)

2.3 Spring

Spring är ett populärt ramverk som utökar Java med en mängd ny funktionalitet. De flesta objekt instantieras inte längre på vanligt sätt i kod, utan kommer att skapas som ’bönor’ av Spring.

Vilka bönor som skapas så definierar man antingen genom en XML- eller Javakonfiguration. Spring använder även någonting som heter Dependency Injection (Beroende injicering) för att hantera beroenden mellan olika objekt. När man skapat en böna så kan den användas som ett vanligt objekt eller vidare injiceras till en annan böna. (Pivotal Software, 2018)

Att injicera en böna i en annan betyder i praktiken att man skapar ett objekt av den och anger det objektet som ett konstruktorargument till ett annat objekt.

I figur 4 visas hur en böna kan skapas när man använder Java-konfiguration.

```
@Bean
public FeeCalculationService feeCalculationService(
        @Qualifier("feeDiscountService") final FeeDiscountService feeDiscountService,
        @Qualifier("feeDomainService") final FeeDomainService feeDomainService) {
    return new StandardFeeCalculationService(campaignDiscountService, feeDiscountService,
            feeDomainService, accountRepository, cardTypeRepository);
}
```

Figur 4: Exempel på hur en böna skapas i Spring

@Bean-annoteringen säger åt Spring att denna metod kan användas för att skapa en böna.

@Qualifier-annoteringen säger åt Spring att söka efter bönor med namnen som anges (t.ex. FeeDiscountService), och skicka dem som argument till metoden.

I figuren ser man även ett exempel på Dependency Injection (beroende-injicering), där de klasser som FeeCalculationService använder sig av, skapas externt av Spring och sedan skickas som konstruktor-argument.

2.4 Metoder

I arbetet kommer jag att gå mera in på olika metoder i Java. En metod (eller funktion) i Java är en isolerad del kod i en klass som erbjuder en viss funktionalitet. Dessa kommer i flera olika tillgångsnivåer, men dom två viktigaste är offentliga och privata metoder.

En offentlig metod kan användas av alla som har en instans av ett objekt, och är den funktionalitet som klassen ska erbjuda åt andra.

En privat metod kan endast användas inom samma klass som metoden själv är i. Dessa används oftast för att dela upp kod i mindre, mer lättlästa block utan att exponera den metoden till andra klasser. En tumregel för privata metoder är att de gärna får vara statiska, d.v.s. att de är oberoende av objektets tillstånd och endast behandlar sina egna parametrar.

2.5 Legacy-kod

”Legacy” är ett ord som används mycket inom IT-världen, och kan beskrivas som ett system som är gammalt och som är svårt att underhålla. Legacy existerar både inom hård- och mjukvara, och i detta arbete så är det mjukvaru-delen som kommer att tittas närmare på.

Legacy-kod kan beskrivas som kod som oftast är gammal och svår att arbeta med. Sådan kod saknar oftast dokumentationen och tester, vilket gör koden svår att förstå. En till orsak varför legacy-kod är svår att arbeta med är brist av kunskap, att det inte längre är någon som vet hur systemet fungerar eller varför det ser ut som det gör.

Detta leder till att det går åt mycket resurser för att underhålla systemet och lägga till funktionalitet till systemet tar mycket längre än vad det borde. (Fischer, 2018)

2.6 Omstrukturering av kod

Att omstrukturera kod går ut på att förbättra en design av gammal kod utan att förändra programmets beteende. Legacy-kod är ett bra exempel på kod som kan vara i behov att att struktureras om. Om man fortsätter bygga på legacy-kod utan att göra något åt designen så kommer koden bara att bli värre och värre.

Figur 5 visar ett exempel på hur dessa problem bygger på varandra.

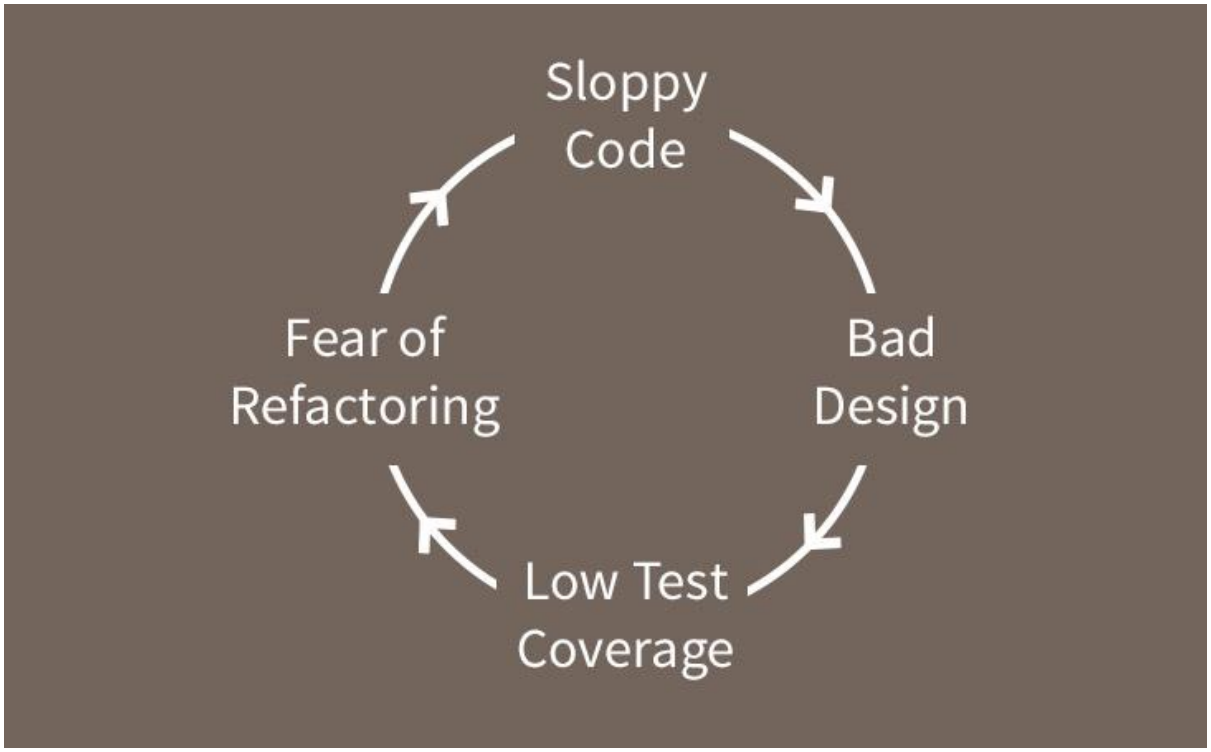


Figure 5: Ond cirkel av dåligt designad kod (Code Climate, 2013)

Figuren visar hur dåligt designad kod oftast saknar tillräckliga tester, vilket leder till att man oftast inte vågar göra något åt koden. Lägg även märke till att vilken som helst av dessa fyra kriterier kan påbörja en spiral som resulterar i att designen blir sämre och sämre.

En viktig del när man ska utföra en omstrukturering är att koden går att testa automatiskt. Då kan man försäkra sig efter varje steg av processen att koden fortfarande utför sin uppgift och att den fortfarande fungerar likadant som tidigare. (DZone, 2018)

Så vad gör man om koden man ska strukturera om saknar automatiska tester? Processen blir aningen svårare. Är det att enkelt skriva tester för koden så är det en bra idé att göra det. Men det lönar sig inte heller att spendera veckor på att skriva tester för en stor del kod. I detta fall så borde man att kommunicera med någon chef eller manager och få reda på vilka användarscenarion som det är tänkt att koden ska sköta om. När man strukturerar om sådan kod så får man komma ihåg att skriva tester för de nya gränssnitt man skapar. Då kan man försäkra sig om att den nya koden är robust och säker. (Code Climate, 2013)

3. OMSTRUKTURERING

Att jobba med legacy-kod kan vara en tidskrävande process. Att skriva om en gammal klass kan gå fort, eller ta väldigt länge, beroende på en mängd olika faktorer, bl.a.:

- Längd – En klass som består av flera tusen kodrader innehåller antagligen mera logik än en som endast är ett par hundra. Den kommer på så vis att kräva både mera förberedande arbete och även flera nya klasser om så behövs.
- Komplexitet – En klass kan ansvara för flera olika områden. En klass som är väldigt lång innehåller ofta logik som skulle passa bättre i en egen klass. Om en klass är för komplex så blir det svårt att förstå vad koden gör eller vilken del av programmet den ansvarar för.

Min omstrukturering gjordes i flera olika steg:

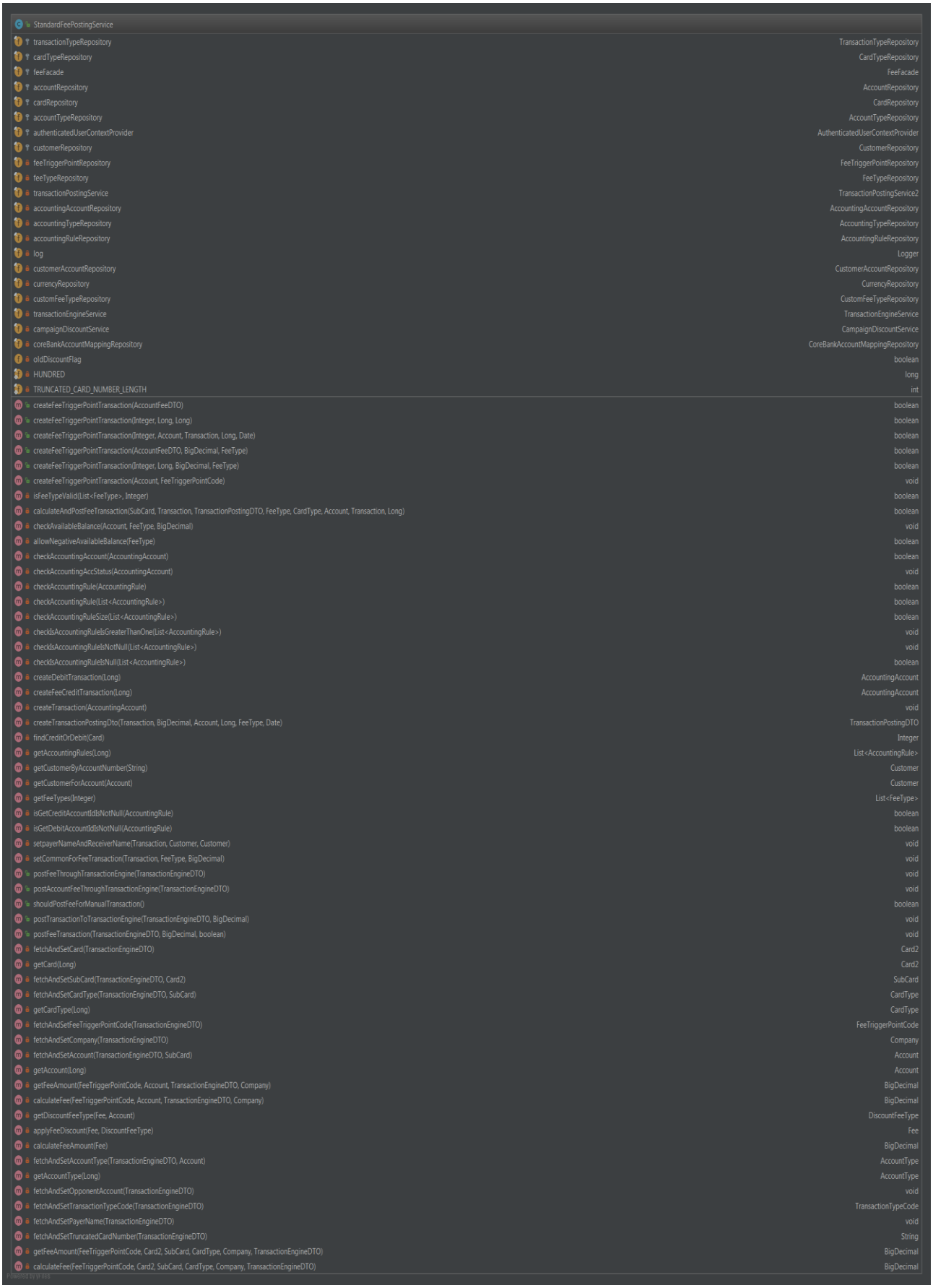
1. Undersöka vilken del av funktionaliteten som inte krävde någon kontakt med andra delar av systemet, och således kan flyttas till en avskild domän.
2. Skapa en ny avgiftsdomän och flytta den koden som går dit.
3. Ta den resterande koden av avgiftshanteringen och dela upp den i flera mindre klasser, och gör den mera läs- och testbar.
4. Skriva automatiserade tester för de nya klasserna.
5. Byta ut användningen av de gamla klasserna mot de nya.

Koden som idag hanterar avgifter så är tätt kopplad till flera olika delar av systemet och klasserna är långa och otydliga.

Den första delen av avgiftshanteringen och den som kommer att behandlas inom detta arbete är den del som hanterar postningen av enskilda avgifter till ett kundkonto. Den funktionaliteten sköts idag av en klass som med åren har blivit onödigt lång och komplex.

Som en konsekvens av klassens komplexitet har den utökats istället för att blivit återanvänd när någon har behövt den.

I figur 6 ses en bild över hur klassen ser ut i dagsläget, med medlemsvariabler och metoder.



Figur 6 FeePostingService (Medlemmar + Metoder)

Figur 6 visar flera tecken på att klassen är i behov av omstrukturering.

- Flera metoder med samma namn, fast med olika parametrar och implementation.
- En uppsjö av (otestbara) privata metoder som är beroende av objektets tillstånd.
- En stor mängd beroenden till andra klasser (visat av de gula symbolerna i Figur 5).

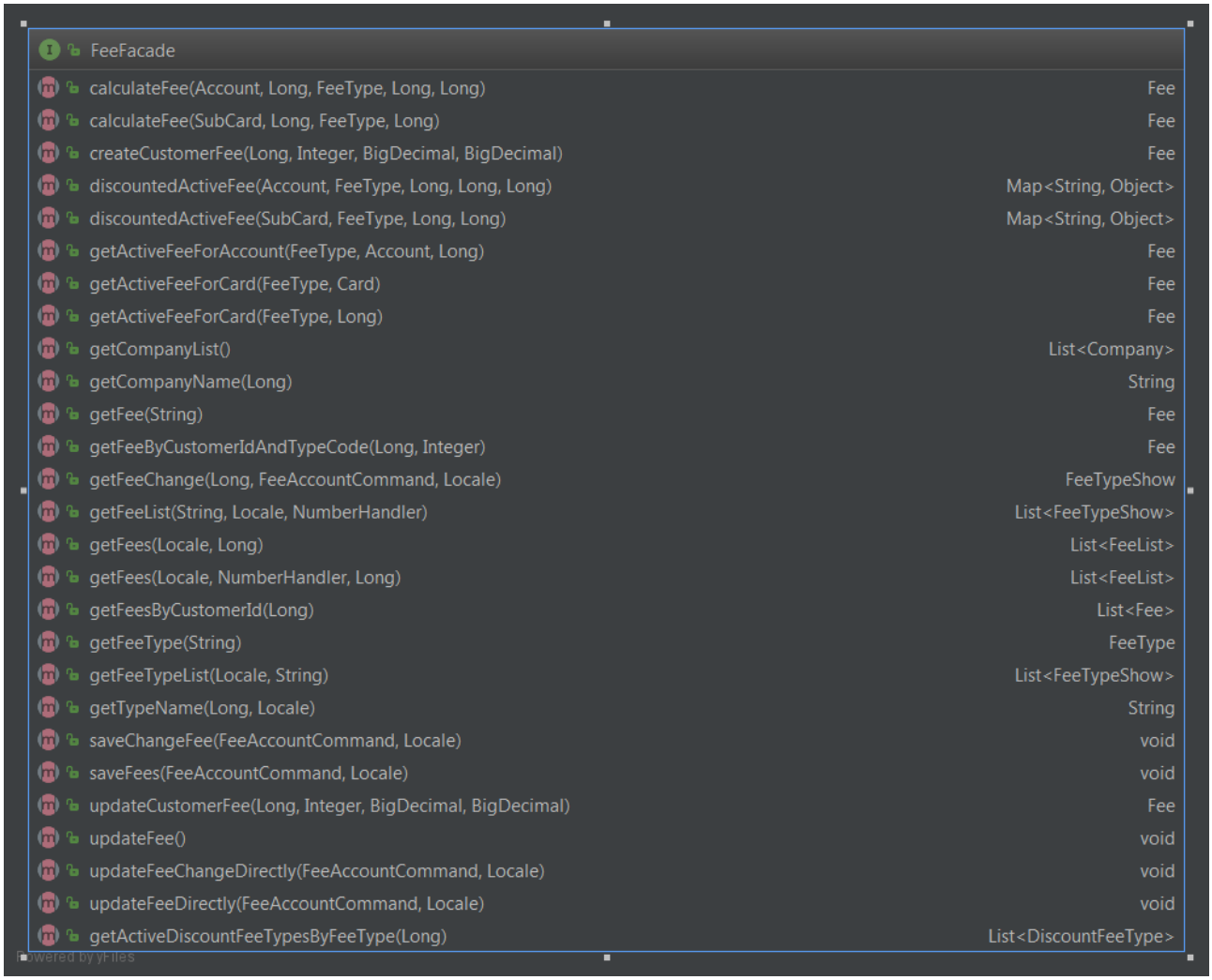
Det framstår ganska klart från figuren också att antalet metoder leder till att klassen i sig har blivit väldigt lång.

En stor del av avgiftshanteringen sköts idag av en fasad över den del av systemet, men arbetet kommer endast att behandla de delar av fasaden som behövs för avgiftspostning:

- Hitta rätt enskild avgift baserat på input-parametrar.
- Räkna ut totalt pris för en avgift (kan vara fast eller procentbaserat pris).
- Applicera en rabatt på en avgift (t.ex. rabatt på uttagsavgifter över en helg).

> ” Facade - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.” (SourceMaking, n.d.)

I figur 7 visas metoderna i FeeFacade.



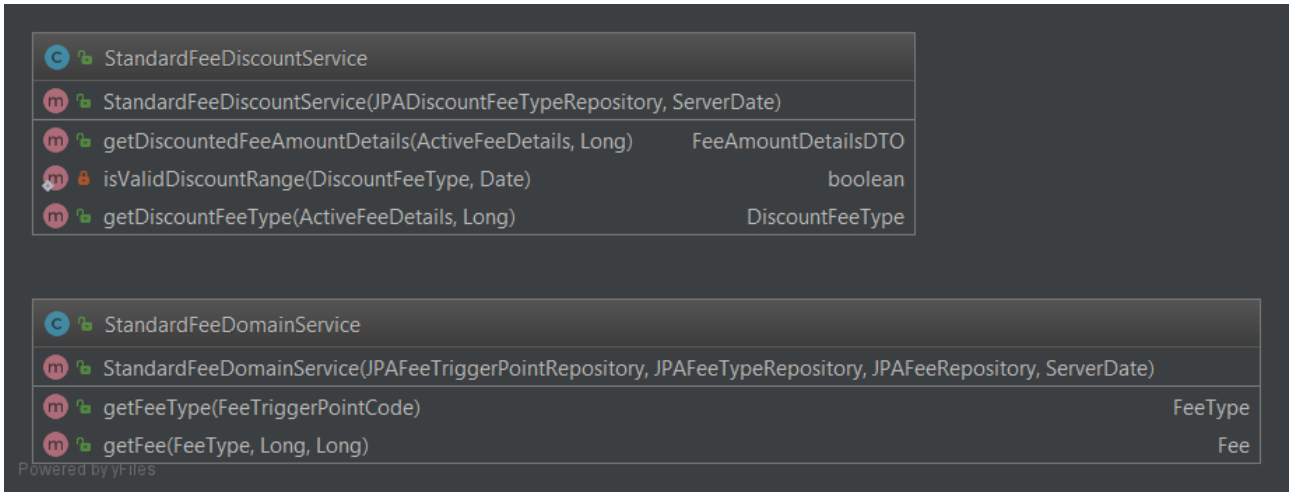
Figur 7 FeeFacade interface

Att påbörja en omstrukturering till en domändriven design från en legacy-kodbas är svårt, eftersom de olika delarna av koden är tätt kopplade till varandra. För att få en komplett domändriven struktur över koden skulle det krävas att flera delar av applikationen följde designen och det går utanför avgränsningarna för detta arbete.

Efter en del undersökande och diskussion med handledare så kunde vi extrahera två nya klasser ifrån våran avgiftsfasad.

- En klass för att hantera rabattering av avgifter.
- En klass för att hantera kopplingar mellan olika domänobjekt, och agera som ett högnivå gränssnitt mot databasen

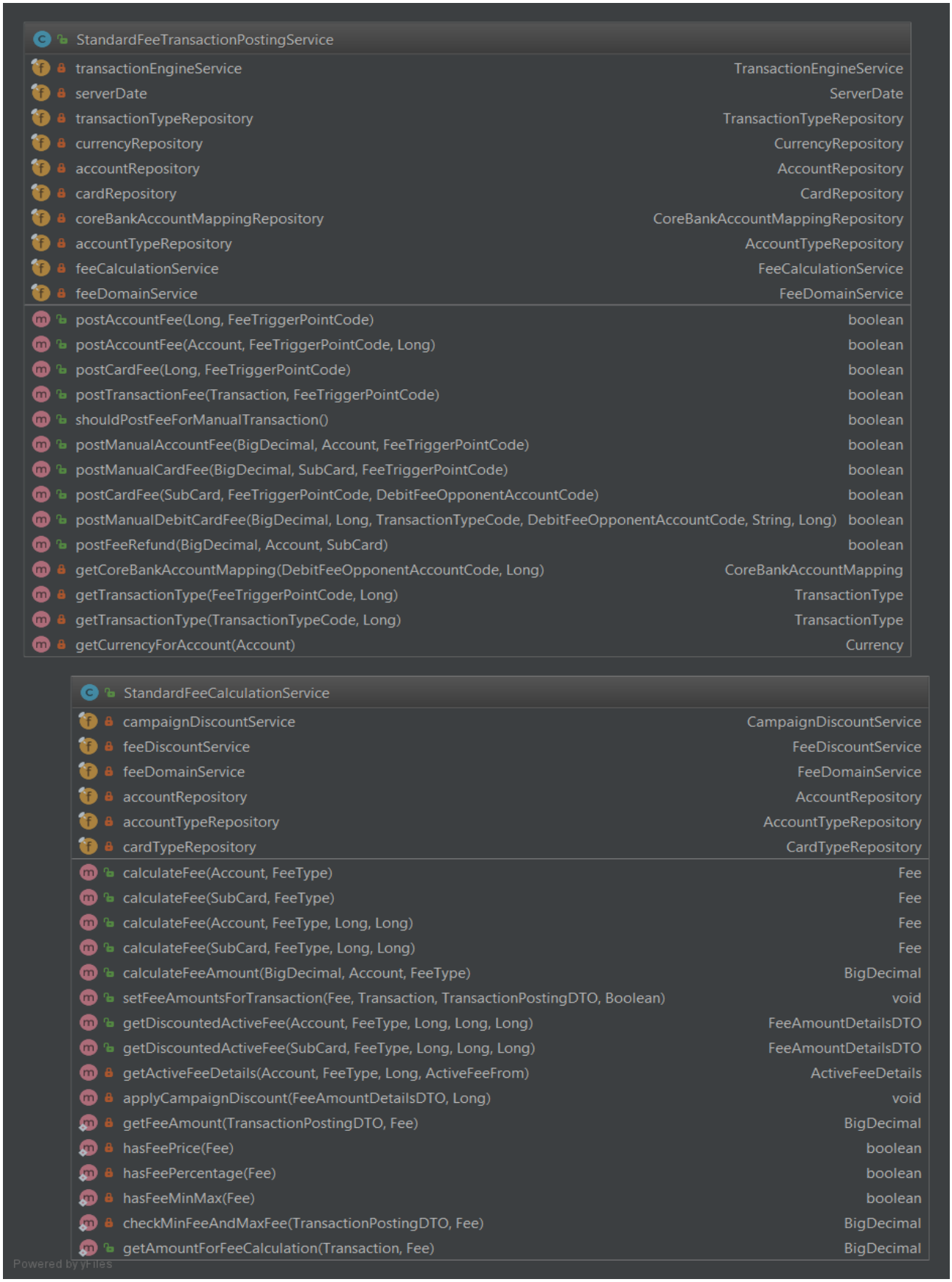
I Figur 8 så visas dessa två nya klasser.



Figur 8: Klasser i den nya domänen

Här är gränssnitten betydligt mindre än tidigare och klassnamnen är lite mer beskrivande. Tyvärr är majoriteten av logiken såpass tätt kopplad till andra delar av systemet så att en stor del av de tidigare klasserna inte kunde placeras i den nya domänen. Så vad gör vi med logiken som inte kunde flyttas till den nya domänen?
Svaret är att den skrivs om till flera mindre klasser, som är mera begränsade i ansvar, och mer testbara än tidigare.

I Figur 9 visas de två nya klasserna som blev i applikationslagret.



Figur 9: Avgiftsklasser i applikationslagret

De nya klasserna är i stort sett kopierade från den tidigare logiken, men har fått ett snyggare och mera lättförståeligt gränssnitt och överflödig eller onödig kod har tagits bort.

Klasserna innehåller fortfarande mycket logik och hade kunnats omstruktureras vidare, men detta skulle kräva att man dök in i flera olika områden avsystemet, vilket skulle gå utanför begränsningarna för arbetet. Det viktiga här är att mängden privata metoder har sjunkit något. Privata hjälpmetoder är bra för att göra koden mera lättläst och enklare att följa, men medför även nackdelar. Den stora nackdelen i mitt arbete är att privata metoder inte går att enhetstesta med konventionella metoder, vilket gör det svårt att verifiera att metoden fungerar som den borde.

4. VAL AV METODER

4.1 Val av databasanslutning

När det kommer till metoder för att ansluta sin applikation till en databas fanns det två möjligheter att välja mellan: Spring Data JPA och MyBatis.

Med Spring Data JPA använder vi oss av speciella klasser för att skapa och använda databasfrågor. Objektet kommer att färdigt innehålla grundläggande metoder för att t.ex. hämta ett objekt utifrån primärnyckel, eller för att spara ett befintligt objekt i databasen. Men det finns även stöd för att manuellt skriva frågor till databasen.

Är man inte i behov av av någon komplex fråga, så kan verktyget färdigt generera de flesta frågor utifrån metodens namn. Men om man behöver, går det även att helt manuellt skriva en fråga genom att annotera metoden med @Query. (Gierke, Darimont, Strobl, & Paluch, 2018)

I Figur 10 visas ett exempel på en sådan metod.

```
@Query("select p from PostalCode p " +
        "join p.countryCode cc " +
        "where p.postalCode = :postalCode and cc.isoCountryCode = :isoCountryCode")
PostalCode findByPostalCodeAndIsoCountryCode(
        @Param("postalCode")final String postalCode,
        @Param("isoCountryCode")final String isoCountryCode);
```

Figur 10: Exempel på manuell query med Spring Data JPA

MyBatis är ett ramverk som också hanterar anslutningar till databaser i Java, där SQL-frågor skrivs manuellt i XML-filer. Förutom SQL-satserna kan man också manuellt skriva så kallade Result Maps (resultat-mappningar). Dessa *Maps* har till uppgift att ta resultatet från en SQL fråga och konvertera det till ett Java-objekt.

Spring Data JPA kräver mindre konfiguration och de flesta metoder fungerar ”Out-of-the-box”, vilket gör JPA till en attraktiv lösning för många.

MyBatis kräver en del konfiguration och manuellt skrivande för att det ska fungera. Det positiva är att när man har konfigurerat färdigt så har man i princip full kontroll över allt som händer, till skillnad från JPA där det mesta sköts ”osynligt” av Spring.

I slutändan valde jag att använda Spring Data JPA. Varför jag valde den är för att den krävde mindre tid att konfigurera, och att jag inte heller hade något behov av att den komplexa funktionalitet som MyBatis erbjuder.

4.2 Val av Spring-konfiguration

I Spring finns det två sätt att konfigurera hur ens bönor ska skapas och hur de injiceras i andra bönor. Detta kan antingen göras i XML-filer eller i dedikerade Java-klasser. XML-versionen används genom att man definierar bönorna med ett namn och vilken klass som ska instansieras, samt eventuella konstruktor-parametrar. Väljer man att använda Java-konfiguration istället så skapar man en klass som annoteras med @Configuration. Då kommer Spring att använda den klassen för att instansiera och injicera bönor. Java-konfiguration brukar även anses vara mer modern och är lättare för en utvecklare att läsa.

Mitt val av konfiguration är Java-konfiguration. Till störst del var det en personlig preferens, då XML snabbt kan bli svårt att förstå sig på.

Spring kan automatiskt scanna efter klasser med vissa annoteringar, och skapa bönor av dem utan att man manuellt behöver skapa dem med en konfiguration.

Figur 11 visar hur en sådan konfiguration kan se ut.

```
@Configuration
@ComponentScan(basePackages = "fi.crosskey.card.fee")
@ImportResource("classpath:/fi/crosskey/card/config/spring/config/general-config.xml")
public class FeeConfiguration {

}
```

Figur 11: Exempel på Java-konfiguration med Component Scanning

I figur 11 visas hur en Java-konfiguration kan skapas. Klassen har tre annoteringar som alla tillför någon slags funktionalitet:

- @Configuration – Klassen känns igen av Spring som en konfiguration och Spring kommer att använda klassen för att skapa bönor.
- @ImportResource – Det går även att i Java-konfiguration importera konfigurationer gjorda i XML och vice-versa. Detta gör det lätt att använda både XML- och Javakonfigurationer, om man så önskar.
- @ComponentScan – Spring kommer att söka igenom det givna paketet efter klasser med särskilda annoteringar och färdigt skapa bönor av dem. Exempel på några sådana är @Repository-klasser som vi tog upp tidigare, men det finns även @Service-annoteringar.

I Figur 12 och 13 så visas exempel på hur man kan markera klasser så att de känns igen av Spring. Figur 13 visar även en speciell klass som används som databasanslutning som jag skrev om i kapitel 4.1

```
@Service("feeDomainService")
public class StandardFeeDomainService implements FeeDomainService {
```

Figur 12: Exempel på @Service-annotering med manuell namngivning

```
@Repository
public interface JPAFeeRepository extends CrudRepository<Fee, Long>
```

Figur 13: Exempel på @Repository-annotering

I båda dessa fall kommer Spring att skapa bönor av dessa klasser, förutsatt att de ligger under det paket som anges i ”basePackages”-parametern. I Figur 12 ser vi även att det går att manuellt ge ett namn åt bönorna.

5. RESULTAT

Resultat för arbetet blev, efter en del avgränsningar, lyckat. En ny domän för avgifter har skapats och en del av den relevanta logiken har flyttats ut, men systemet är ännu långt ifrån en komplett DDD-stuktur. Mycket onödig kod har tagits bort och klasserna är betydligt mera förståeliga. De nya klasserna är även betydligt mer robusta tack vare introduktionen av unit-tester för dem, vilket nästan inte alls fanns tidigare.

- En ny domän för avgiftshantering har skapats, och en del av logiken har extraherats dit.
- Den kod som blev kvar i applikationslagret har blivit uppdelad i flera mindre klasser.
- De nya klasserna är manuellt testade för att försäkra att funktionaliteten är den samma som tidigare.
- De nya klasserna har automatiserade tester.

6. SLUTSATS

Syftet med arbetet är utfört i och med att:

- Metoden formulerad i kapitel 1 har följts när lösningarna har tagits fram.
- Avgränsningarna som nämns i kapitel 1 har följts, likaså nya avgränsningar som kommit upp under arbetets gång.
- Resultaten i detta arbete är de fyra punkterna som är uppräknade i kapitel 5 Resultat.

Arbetet har varit intressant och lärorikt, men har vid flera tillfällen kännits som att man gjort samma sak om och om igen, utan att egentligen komma någonstans. Utöver detta har nya avgränsningar dykit upp på nästan varje vecka, eftersom de flesta delar av arbetet tagit mera tid än förväntat.

Som jag skrev i kapitel 1.2 Metod är detta arbete främst en språngbräda för framtida förändringar. Det finns fortfarande mycket att göra inom domänen och omstruktureringen kommer att fortsätta även efter detta arbete är avslutat.

Den största förändringen som är planerad är att kommunikationen till domänerna/modulerna ska göras via ett REST-gränssnitt, som tillåter maskin-till-maskin kommunikation att ske med hjälp av webbtjänster.

Referenser

Code Climate. (2013, 12 05). *Code Climate*. Retrieved 05 13, 2018, from Refactoring Without Good Tests: https://codeclimate.com/blog/refactoring-without-good-tests/

Crosskey. (n.d.). Retrieved 04 27, 2018, from About Crosskey: https://www.crosskey.fi/our-story/

Evans, E. (2003). *Domain-Driven Design - Tackling Complexity in the Heart of Software.*

Fischer, C. (2018, January 8). Extending Legacy Software with Functional Programming.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code.*

Gierke, O., Darimont, T., Strobl, C., & Paluch, M. (2018, 04 04). *Spring*. Retrieved 04 24, 2018, from Spring Data JPA - Reference: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/

Perry, S. (2017, 8 24). *Introduction to Java programming*. Retrieved 4 29, 2018, from IBM developerWorks: https://www.ibm.com/developerworks/java/tutorials/j-introtojava1/

Pivotal Software. (2018, 04 03). *Core Technologies*. Retrieved 04 29, 2018, from Spring Framework Documentation: https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html

SourceMaking. (n.d.). *Facade Design Pattern*. Retrieved from Design Patterns & Refactoring: https://sourcemaking.com/design_patterns/facade

Vernon, V. (2013). *Implementing Domain-Driven Design.* Addison-Wesley.

VersionOne. (n.d.). *VersionOne | Unified Agile & DevOps*. Retrieved 05 13, 2018, from Code Refactoring in Agile Programming: https://www.versionone.com/agile-101/agile-software-programming-best-practices/refactoring/